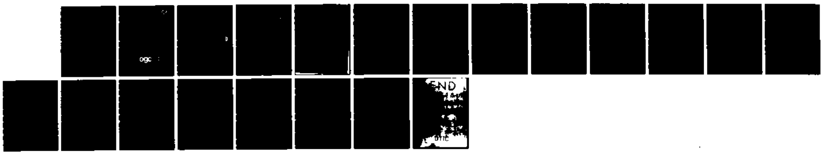
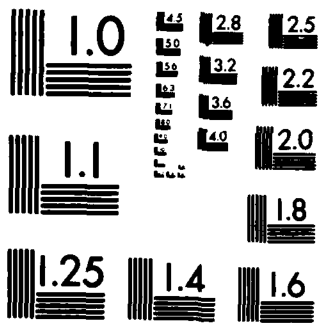
AD-A147 104 DEBUG TESTING AND CONFIDENCE TESTING(U) MARYLAND UNIV 1/1
COLLEGE PARK DEPT OF COMPUTER SCIENCE D HAMLET AUG 84
CS/E-84-004 AFOSR-TR-84-0867 F49620-80-C-0004

UNCLASSIFIED F/G 9/2 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

12

*Technical Report CS/E 84-004 August, 1984*

# DEBUG TESTING AND CONFIDENCE TESTING

Dick Hamlet
*Oregon Graduate Center*

ogc
OREGON GRADUATE CENTER

19600 N.W. WALKER ROAD
BEAVERTON, OREGON 97006

[illegible]

*Technical Report CS/E 84-004 August, 1984*

# DEBUG TESTING AND CONFIDENCE TESTING

Dick Hamlet
*Oregon Graduate Center*

## Abstract

The strong point of computer program testing has always been failure. When a test fails, it is clear what to do, and this has led to the maxim that the goal of testing is finding faults. Testing theory, on the other hand, has tried to connect test success to program correctness. Call the kind of testing that seeks failures '*debug testing*,' and the other '*confidence testing*.' A confidence-testing technique might in principle be used for debugging, but debugging tools cannot establish confidence. Debug testing is an activity intertwined with the whole of program development, and its theory must take account of this sociological context; debugging is a human craft. On the other hand, confidence testing theory may take program and test as given, without their human origins. Only by separating the two kinds of testing can reasonable goals be set for testing theory.

The difference between debug- and confidence-testing theory is illustrated by detailed analysis of partition testing, and of experiments to validate debugging test tools. Goals for each kind of theory are proposed.

*Index terms*: testing theory, debugging, partition testing, nose-rubbing effect, and probabilistic correctness

This work was partially supported by the Air Force Office of Scientific Research, under grant F49620-80-C-0004.

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br><br>2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited. |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>TR CS/E-84-004 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>AFOSR-TR- 84-0867 |

| | | |
|---|---|---|
| 6a. NAME OF PERFORMING ORGANIZATION<br>University of Maryland | 6b. OFFICE SYMBOL *(If applicable)* | 7a. NAME OF MONITORING ORGANIZATION<br>Air Force Office of Scientific Research |
| 6c. ADDRESS *(City, State, and ZIP Code)*<br>Department of Computer Science<br>College Park MD 20742 | | 7b. ADDRESS *(City, State, and ZIP Code)*<br>Directorate of Mathematical & Information Sciences, AFOSR, Bolling AFB DC 20332 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>AFOSR | 8b. OFFICE SYMBOL *(If applicable)*<br>NM | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>F49620-80-C-0004 |
| 8c. ADDRESS *(City, State, and ZIP Code)*<br>Bolling AFB DC 20332 | | 10. SOURCE OF FUNDING NUMBERS |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| 61102F | 2304 | A7 | |

11. TITLE *(Include Security Classification)*
DEBUG TESTING AND CONFIDENCE TESTING

12. PERSONAL AUTHOR(S)
Dick Hamlet*

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT *(Year, Month, Day)* | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM ______ TO ______ | AUG 84 | 16 |

16. SUPPLEMENTARY NOTATION
*Now with the Dept of Computer Science, Oregon Graduate Center, Beaverton OR 97006.

17. COSATI CODES

| FIELD | GROUP | SUB-GROUP |
|---|---|---|
| | | |
| | | |

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Testing theory; debugging; partition testing; nose-rubbing effect; probabilistic correctness.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The strong point of program testing has always been failure. When a test fails, it is clear what to do, and this has led to the maxim that the goal of testing is finding faults. Testing theory, on the other hand, has tried to connect test success to program correctness. Call the kind of testing that seeks failures debug testing, and the other confidence testing. A confidence-testing technique might in principle be used for debugging, but debugging tools cannot establish confidence. Debug testing is an activity intertwined with the whole of program development, and its theory must take account of this sociological context; debugging is a human craft. On the other hand, confidence testing theory may take program and test as given, without their human origins. Only by separating the two kinds of testing can reasonable goals be set for testing theory.

The difference between debug- and confidence-testing theory is illustrated by detailed analysis of partition testing, and of experiments to validate debugging test (CONTINUED)

| | | |
|---|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>[✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Dr. Robert N. Buchal | 22b. TELEPHONE *(Include Area Code)*<br>(202) 767- 4939 | 22c. OFFICE SYMBOL<br>NM |

DD FORM 1473, 84 MAR    83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ITEM #19, ABSTRACT, CONTINUED: tools. Goals for each kind of theory are proposed.

## 1. The Two Kinds of Testing

Programs are tested to investigate their behavior, but with two distinct purposes. During debugging, programs are *not* correct, so good tests expose their faults, and lead to understanding. Debug testing is a subject whose theory is mostly sociological: the methods and quirks of programmer and process are important. In particular, programmers design and code from specifications, and the faults in a program can often be traced to misunderstandings at the specification-program interface. With a few exceptions, practical "testing systems," notably those that use coverage metrics, are intended to aid in debug testing. Myers [1] has given the best advice for the debug tester: the faults are there--look for them. Theoretical investigation of debug testing is properly supported by experiment, and these experiments are notoriously difficult to do, because the entire programming process is involved, and what works in practice may critically depend on an obscure facet of the process. Thus we should not expect to generalize beyond the precise situation of the experiment. Such studies cannot make predictions until the whole of development is better understood, at least enough to know what variables to control. And to hope for a theory free of the sociological component, a hard science rather than a soft one, is probably fond. Program development may not be an art, but it is at least a craft, and so long as people do the development, debugging the results will depend on the human process. Debug testing and validation experiments are discussed in Section 2.

The second kind of testing follows debugging, and is intended to establish confidence in the program's correctness. So long as tests are exposing faults, looking for faults is a good plan. When all the tests succeed, debugging is evidently over, but no guide such as Myers's is then available to continue. However, the theoretical situation in confidence testing is much cleaner. The program and its tests are well-defined objects, whose relationship to each other need not depend on the human process that created them. A number of test criteria have been developed to capture the idea that success establishes confidence in correctness. Almost all such measures lead to unsolvable problems, rooted in the problem of deciding whether two arbitrary programs have equivalent functional behavior. However, it is not very satisfactory to define a good test only to show that good tests cannot be generated or recognized algorithmically. It is more promising to view tests as samples drawn from a program's behavior space, and to make probabilistic calculations of the program's quality based on the samples. Probabilistic theories get the right *kind* of answers, but they are quantitatively wrong: the question of sample independence seems to be the root of the trouble. The present failings of confidence testing are discussed in Section 3.

There is considerable confusion between debug testing and confidence testing. If one had a method that established confidence in a program's correctness, it could certainly be used for debugging, as follows:

> Tests selected for a buggy program would either expose bugs, or predict low confidence in their absence. In the first case the program needs work; in the second, the test should be improved. In the process, bugs will be eliminated and confidence will increase until the software can be released.

Unfortunately, there are no confidence-testing methods that can be used in this way, because there is no theory that establishes a solid connection to correctness yet does not founder on undecidability. What happens instead is the following:

Debug testing is done according to some scheme whose implications for program correctness are entirely unknown. The testing is successful in that bugs are found and repaired; eventually, no more are found despite considerable effort by the testers. The composite debug test is then thought to establish confidence in the program.

It is obvious that this practical method is indefensible so long as the relationship between the methods used and the program's correctness remains unknown.

As an example of the confusion between debug and confidence testing, a detailed analysis of partition testing is given in Section 4. Its debug component relies on a part of the process by which people code from operational specifications; its confidence component is a proof method involving no testing at all.

The goal of debug-testing theory should be to explain why, in terms of the human process by which software (and bugs) are created, that a method locates faults. Some way must be found to eliminate the distorting effect of human cleverness on these explanations. It is equally important to explain the weak points of a method--the kind of bugs that it likely fails to find. The goal of confidence-testing theory remains that of finding the relationship between a successful test and program correctness. A probabilistic theory will be required, in which the question of test independence is central. Goals and suggestions for future research are given in Section 5.

## 2. The Sociology of Debug Testing

In the absence of a theory that relates test success to program correctness, many excellent test systems have been devised for debugging. By far the most common tools are the path analyzers, which report or generate test coverage of the control structure [2, 3]. A test that fails to cover some path is evidently useless to uncover a fault on that path. On the other hand, the significance of covering all paths is unknown, beyond the general result that such coverage is not logically related to correctness. Mutation [4, 5], another structural-coverage technique (for expressions), is only a bit better: when coverage is less than perfect, bugs could evade the test; when coverage is perfect, we know only that some particular bugs are absent. (However, this is the origin of a good idea, *fault-based testing*, that does have some real theory. By restricting attention to particular faults, the theory can sidestep the sociological question of where those faults came from, and concentrate on the technical problems of surely finding them [24, 25].) Different structural coverage techniques are difficult to compare. Some are obviously subsumed by others (statement coverage by branch coverage, for example); some appear to be unrelated (e.g., mutation coverage and path coverage) [6, 7]. In keeping with the practical nature of debug-testing schemes, it is easy to contrive a useful one that defies even comparative analysis (for example, the variation of path coverage [8]).

Programming and debug testing are linked by the involvement of clever, dedicated people who do both. Experiments conducted to establish or compare the efficacy of debug-testing methods can be entirely misleading, because they fail to control for this human involvement, and for the very special form taken by programs built by people. A good name for the distortion introduced into a testing experiment by these factors is the "nose-rubbing effect."

Consider the simplest control coverage criterion, that every statement of a program must be used in some test. The conventional explanation for why statement-coverage testing works is that the coverage leaves bugs no place to hide. But this explanation ignores the human analysis that goes into finding tests that attain coverage. An unexecuted statement is examined to see what it does, and this analysis may well uncover some fault in it. Since it has not been executed, the credit for finding this fault certainly goes to the person, not the test method. Once a missed statement's purpose is understood, other parts of the software must be analyzed to see why it was missed, and this requires examination of paths and predicates that might lead to it. In the process the tester looking for trouble is liable to find it, but not necessarily trouble related to the unexecuted statement. (Perhaps for an omitted case that uses this statement, the software works perfectly.) In the "nose rubbing" process a technical expert, looking for trouble, is forced to examine particular parts of the code, and there (or nearby) faults are found. Code being what it is, close study almost always finds bugs.

It might be instructive to conduct a study of pure nose rubbing. Programmers would be told that they were evaluating a new testing tool, and would be given programs containing faults. When these were executed under control of the "tool," an arbitrary pointer into the code would be printed with an obscure error message. This message would be removed by a change in the program that repaired any fault, and another generated, until all the known faults were removed. Positive results from such a negative experiment would cast doubt on naive "evaluation" of testing tools.

Howden [9] has conducted a different kind of empirical study: he applied path analysis to a few small programs to find its worst-case performance. He counted the method a success only when the human contribution was eliminated--no matter how the paths were covered, the fault led to an observable failure. An important result of this study was the difficulty of performing the analysis (necessarily by hand)--the technique cannot be used on even moderate-size programs, so it is unlikely that the study will be extended. The numerical results (that path testing necessarily would expose about 65% of the errors in 11 toy programs from a textbook) cannot be trusted as an evaluation of path testing.

The sociological theory of debug testing must take into account the program development process, and explain the efficacy of bug-finding methods in terms of the human crafts of design from specification, coding from design, and testing a program against its specification. As we learn more about the ways people carry out these tasks, we will learn better debugging methods and why they expose the mistakes people make. An example of such an analysis is given for partition testing in Section 4.1.

## 3 Confidence Theory and its Failings

The initial attempts to understand why tests seem to improve the confidence that a program contains no more faults were related to program correctness. The failure of this "absolute" theory has more recently led to the use of probabilistic ideas.

## *3.1 L[illegible] D[illegible]tions and Negative Results*

In o[illegible] to speak precisely, we adopt the functional semantics of Mills [10, 11].

**Definition.** The meaning of a program $P$ is the mapping from inputs to outputs it computes; this functional meaning is written $\boxed{P}$. If $P$ executes successfully on input $x$, $\boxed{P}(x)$ is its output. A *specification* $f$ is a computable function that $P$ should compute. A *test* $T$ consists of a set of input values for $P$. Since testing is actually to be carried out, $T$ is finite, and $\boxed{P}$ is defined on $T$. A test is *successful* when each input in $T$ meets the specification. That is,

$$\forall t \in T\,(\boxed{P}(t) = f(t)).$$

A specification is *effective* iff there is a mechanical way of deciding if any given test is successful. Program $P$ is *correct* (with respect to specification $f$) iff $f \subseteq \boxed{P}$.

This definition allows the program-function domain to be larger than that of the specification–intuitively, by failing to specify what the program should do, no one means to *require* it to blow up.

The most important definition in testing theory must capture the relation between success of a test and correctness of the tested program. Such tests are what we are seeking. Howden's definition at first seems to be a direct translation of the intuitive idea:

**Definition.** A test $T$ is *reliable* (Howden [9]) iff it cannot succeed without $P$ being correct:

$$T \text{ successful for } P \text{ (with respect to } f) \Rightarrow P \text{ correct.}$$

A basic negative result of testing theory is that the problem of deciding if an arbitrary test is reliable, is unsolvable.

**Theorem**: There is no algorithm for deciding of program $P$ and test $T$ whether or not $T$ is reliable for $P$.
**Proof**. By reduction to the program equivalence problem; see [9, 12, 26].

(It is disturbing that the specification enters these proofs in the form of a program of some sort, indicating that including the idea of correctness forces any imagined algorithm for deciding reliability to be imprecise--for how can the specification be "given" to the algorithm?)

In one pathological case a reliable test does not exist: when program and specification disagree only because the program blows up where it should not. The points not in the domain of the program function cannot be part of any test by our requirements.

Howden's definition has been criticized [13, 14] because it does not capture an idea important in program maintenance: a trivial program change can turn a reliable test into an unreliable one. If a program *is* correct, then any test–even the empty test–is reliable. Thus reliability is attained by perfecting the program, making test points superfluous. This is not at all the intuitive idea that the test is augmented until it catches all the program faults.

Perhaps it better captures a test expanding to force correctness of P to say that if the test is successful for any program, that program must be indistinguishable from P on the specified domain. This gets at the idea that a test is lacking if any failure can escape it.

**Definition:** A test T for program P is *valid* with respect to specification f iff for each program Q,

$$T \text{ successful for } Q \text{ (with respect to } f) \Rightarrow \boxed{Q}\,|_{\text{dom } f} = \boxed{P}.$$

The definition does not state that P must be correct, or even that T is successful for P, but we can prove this.

**Theorem:** If there exists a valid test for P with respect to f, then P is correct (and hence any test is successful for P).
**Proof**. Since f is computable, let program Q compute it, $\boxed{Q} = f$. Suppose that T is valid for P. Because Q computes f, any test is successful for Q, including T, so $\boxed{Q}\,|_{\text{dom } f} = \boxed{P}$. But Q computes f, so $\boxed{Q}\,|_{\text{dom } f} = f$, and hence $\boxed{P} = f$ and P is correct.

**Theorem:** A valid test is reliable, but a reliable test need not be valid.
**Proof**. Suppose T is valid for P. Since P is correct, any test is reliable, in particular T. On the other hand, consider a correct program P with nonempty domain D and the empty test $\phi$, which is reliable. $\phi$ is not valid, because there is a program Q which differs from P on D, yet $\phi$ is successful for Q, violating the definition.

The theoretical situation for valid tests is even worse than that for reliable ones: the only valid tests are exhaustive. This result captures the sense of Dijkstra's aphorism that tests can never demonstrate the absence of faults [15].

**Theorem:** A test is valid for program P with respect to f iff it includes dom f.
**Proof**. It is obvious that an exhaustive test is valid. On the other hand, suppose that point z $\in$ dom f is omitted from test T. Then construct Q to be exactly the same as P except that Q contains an initial test for z which makes their outputs differ at that point alone. Then T is successful for Q, but since $\boxed{Q}(z) \neq \boxed{P}(z)$, T is not valid.

Since tests must be finite sets according to the definition above, only a program for a specification with finite domain could have an exhaustive test. The requirement that the program not blow up for any test point causes less trouble than for "reliable": although determining a program's behavior on arbitrary inputs is an unsolvable problem, human ingenuity would be used to master it on a given domain.

("Reliable" and "valid" have another sense in some literature, notably [16]; however, the criticism of [17] puts the other definitions out of circulation.)

The definition of "test" itself is chosen to make testing algorithmic--tests can in principle always be performed. It is desirable that a testing method--a scheme for judging test points--have a similar property.

**Definition:** A *testing method M* is a three-place predicate. *M* (f, P, T) is defined to hold when the program P with specification f subjected to test T has been tested according to the method. A testing method is *algorithmic* iff *M* can be mechanically evaluated.

This definition is imprecise in that the form in which a specification might be supplied to the predicate is difficult to imagine. However, so-called "structural" methods ignore the specification except to require successful tests. Then the difficulty is hidden by assuming an effective specification. For example, branch testing has the predicate $M_B$(f, P, T) that is true iff T causes each branch of P to be executed, and T is successful for P. Branch testing is algorithmic for an effective specification, because it is easy to monitor execution of all branch points. Similarly, mutation testing is algorithmic if equivalent and long-running mutants are defined to be stillborn. (This assumption cannot be realized in practice.)

If "reliable" and "valid" are taken as testing methods, the first cannot be algorithmic by the theorem that reliability is an undecidable property; the second is algorithmic for finite-domain specifications by the theorem that an exhaustive test is valid (with the heavy proviso that the specification domain be given). Algorithmic methods do not in general attain either reliability or validity. Validity requires an exhaustive test of a given finite specification domain. For reliability things are a little more difficult.

**Theorem:** No algorithmic testing method can always have reliable tests.
**Proof.** Suppose an algorithmic method with predicate *M* were available, such that whenever *M* (f, P, T) holds, T is reliable for P. Then to decide reliability of an arbitrary test T′ for program P, find by trial a set T such that *M* (f, P, T). Success on T means that P is correct. In that case T′ is also reliable, since any set is. On the other hand, if T is not successful, then T′ is reliable iff it is *not* successful. In any case, the reliability of T′ has been determined, contrary to the theorem above. Hence there can be no reliable algorithmic test method as supposed.

Non-exhaustive algorithmic methods are therefore not useful for confidence testing, insofar as the ideas of "reliable" and "valid" capture the connection between testing and correctness.

## *3.2 Probabilistic Theories*

The analogy between program testing and quality-assurance for mass-produced goods is extremely attractive. When the final product of an assembly line is to be checked for defects, the only sure method is to inspect each item. Since this is impractical, samples are inspected, and the confidence that the sample predicts the quality of an item chosen at random can be obtained from statistical theory. In the analogy, the line samples correspond to program tests, from which we would like to calculate the confidence that no faults exist, that is, confidence in the success of arbitrary program executions as yet untried.

The analogy must be faulty, however, because the results from the simplest theory are radically wrong both qualitatively and quantitatively. In the situation that *N* test points chosen at random succeed, and one requires 1 - $\alpha$ confidence that the probability of correctness is $p$, the number of points required [29] is

$$N = \log \alpha / \log p.$$

The qualitative error is that this result is independent of the form and size of the program being tested. Experience has shown that the fault count is roughly proportional to program size [28], and at least for programs with many independent paths, $N$ must then increase with the program size. The result is also independent of the domain size, and random testing intuitively should be less effective for large domains. Quantitatively the formula is also intuitively wrong, since (for example) it predicts that 45 test points are sufficient to establish 90% confidence that the correctness probability is 0.95.

Before we analyze the failure to correspond with intuition, it should be noted that the simple probabilistic theory gives just the right *kind* of statement. It predicts the result of executing the tested program, and does so entirely in terms of information about the test and program. The quantitative results could be used to decide if the program is ready to be released.

The most obvious place to question the quality-control analogy is in the selection of "random" samples. When products come off an assembly line, all are equal; but some program inputs are more equal than others. Each program has an "operational distribution" characteristic of its real use, which weights input classes unequally. The simplest statistical theory demands that to predict confidence bounds for executions drawn from such a distribution, tests must be drawn from it as well. When the operational distribution is unknown (as it must be for a new application) the theory cannot be used, because far more points might be required to cover the real distribution, drawing tests from the wrong one. Furthermore, input distributions do not help to explain the failures mentioned above: using the correct distribution it is *still unreasonable that the required test size does not depend on* the program or domain size, and the numbers are still wrong.

What does it mean for samples from a manufacturing process to be independent? The essential feature is that there be no correlation between defect-producing operations and the sample selection. For example, choosing every $N$th item from a line seems reasonable until it is noted that (say) exactly $N$ components arrive at some workstation in a group--the samples then might all contain the first component from the group, which might be special in some way. In the testing analogy real independence is very difficult to obtain. Tests do not penetrate programs in any uniform way, so each actually "sees" very little of a large program. Furthermore, many tests may see the same fragment, for all that they were not selected to do so. Program fragments are not uniform: they are produced by different people, under different circumstances, to different standards. To attempt to predict the quality of one by examining another is obviously foolish. To turn the analogy around, if manufacturing quality assurance were like program testing, samples would be examined only in part, that part depending on the choice of sample in a way that might correlate with defects.

This way of looking at quality control also clarifies the role of test weighting distributions. In quality control the analogy would be to examine only scattered parts of each sample, and concentrate on some particular parts. This would subvert the whole purpose of quality control, which is to discover the source of defects and eliminate it. The inspection would not be determining that the item was being made correctly, but only that its defects were not glaring. In the case of programs, testing from an input distribution investigates not correctness, but how uncommon failures are.

To correct the flaws in sampling theory, some kind of code dependence must be added, so that samples cover not only the input space, but the textual program space as well.

## 4 Analysis of Partition Testing

Because the idea of correctness comes from program proving, the first testing theories had the same origin. Proof-based theories might be described as program proof methods that incorporate tests. It has long been thought that test success might simplify program verification [18]. For example, for a particular program, it might be possible to construct a proof of correctness that was based on clever (non-algorithmic) choice of a reliable test. A person would prove that the test was reliable, then the test would be conducted, and its success would complete the proof [19]. Or, a person might prove that certain faults in a program would necessarily lead to failures, then establish their absence by conducting a successful test [24]. These general methods cannot be criticized, but they do not lead to a theory of confidence testing. Instead of tests, they really analyze programs.

Beginning with [16], proof-based testing theory has become identified with the much narrower class of methods called "partition testing," in which the program's input domain is broken up into equivalence classes, and test points are selected to cover these classes. The essential idea is that the equivalence classes should be "treated the same" by the program and the specification. This notion has been used in a fault-based theory [17] and in work based on symbolic execution [21, 22]. We now analyze "treated the same" partitions first as a confidence-testing idea, then as a debug-testing idea.

### *4.1 Partition "Testing" is a Program-proving Idea*

Testing based on input partitions for identical output does lead to correctness, as the following trivial theorem shows. Define the same-output equivalence relation for a program $P$ as

$$P^{\ominus} = \{(x, y) \mid \boxed{P}\,(x) = \boxed{P}\,(y)\}.$$

Define the same-output equivalence relation for a specification $S$ as

$$S^{\ominus} = \{(x, y) \mid S(x) = S(y)\}.$$

These relations define partitions of the input space, whose intersection classes have members that are literally treated the same. Members of such a class are all specified to have a single output, and furthermore do have a single output when supplied to the program. In *diagonal* partitions the specified and actual output is the same; in *off-diagonal* partitions the outputs differ.

Of course, if partitions are actually to be the source of tests, they must be of finite index. However, if the definition of a test is relaxed to allow "infinite" input sets, the results of this section continue to hold. The case in which the program domain is smaller than the specification domain deserves special comment. Here there exist inputs for which the program blows up, but should not. Such inputs occur in no intersection partition, because they are not in any partition of $P^{\ominus}$. It is the primary virtue of tests arising from specifications that these inputs not be lost, and this can be arranged by adding an "undefined" partition to those of $P^{\ominus}$ but not to those of $S^{\ominus}$. This creates off-diagonal intersection partitions for failure of definition in P.

**Theorem**: A test using an arbitrary element from each intersection partition of the $P^{\ominus}$ and $S^{\ominus}$ relations is successful iff P is correct with respect to S.
**Proof**. (Correctness as a consequence of test success.) Each of the non-diagonal partitions must be empty for the test to succeed, because by definition P is in error for all points therein. Consider then any nonempty diagonal partition D, and any $x \in D$. Some $t \in D$ was involved in the successful test, and hence $\boxed{P}(t) = S(t)$. But by definition of D, $S(x) = S(t)$ and $\boxed{P}(x) = \boxed{P}(t)$, hence P is correct, because x was an arbitrary input. (The reverse implication is trivial.)

The proof shows that there is an easier way to state this result:

**Corollary**: The off-diagonal intersection partitions of $P^{\ominus}$ and $S^{\ominus}$ are empty iff P is correct.

The obvious practical deficiency here is that the $P^{\ominus}$ and $S^{\ominus}$ partitions are seldom of finite index. For testing, however, the method cannot be used at all, because representatives of the intersection partitions cannot be obtained in practice. Consider the "triangle problem" for example. Triples of integers (A, B, C) representing triangle sides are to be classified into the textbook types such as "obtuse scalene." The possible outputs are a finite set, and thus $S^{\ominus}$ determines a natural finite-index input partition. The natural program that solves the problem has a path corresponding to each possible output, so its path equivalence classes are the partitions of $P^{\ominus}$, also of finite index. Choosing a point from a specification partition like "equilateral" may be easy, and a successful test execution shows that the intersection with the "equilateral" program partition is not empty. But it does not prove correctness to proceed in this way, because there is no way to select points in off-diagonal partitions (such as: specified "equilateral" but the program prints "right isosceles")--indeed, the Corollary states that these partitions must be empty for correctness.

Thus in its simplest form, use of treated-the-same input classes is a proving technique that makes no use of testing at all: the off-diagonal partitions must be shown to be empty, necessarily without testing; then there is no need to try points in the diagonal partitions.

There are natural input partitions for specifications and programs broader than those of $S^{\ominus}$ and $P^{\ominus}$. If these are of finite index, or have easy-to-find representatives, they are candidates for a proving method; however, the same argument shows that such partitions are not useful in a confidence testing method. For example, suppose a first-order logic specification is of the form

$$
\begin{array}{ll}
I_1(x), & O_1(x,y) \\
I_2(x), & O_2(x,y) \\
\dots & \\
I_n(x), & O_n(x,y)
\end{array}
$$

where the $I_i$ are disjoint input assertions, and the corresponding $O_i$ are output assertions for those inputs. Let I describe the specified domain:

$$I = I_1 \vee I_2 \vee \dots \vee I_n,$$

and

$$S_i = \{x \mid I_i(x) \wedge \exists y\, O_i(x, y)\}$$

for each $1 \leq i \leq n$ are input partitions for each part of the specification. (The inputs not in any partition are those for which the specification fails to constrain the result at all, because $\neg I$ holds; and, those for which the specification asks the impossible, because there do not exist outputs as required.)

If a program P is to meet this specification, similarly let

$$P_i = \{x \mid \boxed{P}(x) = y \wedge O_i(x, y)\}$$

and add

$$P_{n+1} = \{x \mid x \notin P_i, 1 \leq i \leq n\}$$

to cover the problem with failures described above. Intersecting specification-defined and program-defined partitions, we obtain classes that are "treated the same" in a wider sense than that used above. The same results hold and there is the same difficulty in test selection. The off-diagonal partitions are now those for which some input assertion $I_k$ holds, but either $\boxed{P}$ is undefined or P's output fails to satisfy $O_k$; these are not easy to identify. If they can be shown to be empty, the proof of correctness is complete without recourse to tests.

This analysis shows that the idea of "treated the same" partitions cannot be used for a confidence testing theory. Correctness turns on empty error partitions that are difficult to identify, and unrelated to successful tests on other partitions. With partitions of infinite index, the non-diagonal number is also infinite, and concentrating on diagonal partitions [22] does not lead to either a proof of correctness or a confidence test.

### *4.2 Partition Testing for Debugging*

As a debug-testing method, partition testing is valuable, but because of the particular way in which human beings create programs from specifications. The great virtue of any test based on a specification is its potential for detecting missing program logic. It is a common programming blunder to omit cases that the specification requires. Because the chance of coincidental correctness is small when part of a program has been omitted, any test point in the specification partition will expose the fault. Furthermore, narrow input partitions which have both a specification- and program-based meaning are good intellectual tools for debugging. When a failure occurs for some input, its characterization both locates the fault (from the program-based partition) and indicates what should be done (from the specification-based partition).

It is an open question whether program- and specification-based partitions should be similar (for example, when an operational specification is taken to be prescriptive), or intentionally different (for example, when a declarative specification technique is used with conventional programs). In the first case the intersection partitions for a correct program differ little from the partitions before intersection; in the second case nothing forces this to be true. Partition boundaries are widely held to be important, because in programming it is common to blunder by shifting boundaries. When this happens the boundary becomes an off-diagonal intersection partition. But boundaries have this intuitive significance only when the specification is prescriptive. Otherwise input distinctions that "should" be made have no significance. For example, for the absolute value function it is common to specify the behavior differently for positive and negative inputs. But the programmer who writes

```
real procedure absval(x); real x; value x;
 absval := sqt(x↑2)
```

is not observing the positive-negative distinction, and separating those classes will not help to find bugs for this program. When specification and program partitions are very different, as in the DAISTS system [30] using data-type axioms and conventional programs, it is difficult to characterize and think about the intersection partitions. It may even be difficult to find points in the partitions [23]. The precision of partitions does not suffer, however, and the distinct nature of specification and program makes bug repair easier. In compensation for lost intuitive understanding of partitions, the nose-rubbing effect is given maximum play: the programmer asked to cover an incomprehensible partition is led to study a narrow part of the specification and program, and probably to discover bugs there.

The very properties that make partitions useless for confidence testing are advantages in debugging, particularly when specification and program partitions are similar. For example, in [22] specification and program are both procedural, and partitions are based on path classes in each. The specification classes thus have nothing to do with correct behavior, and only serve to distinguish arbitrary cases. Within such partitions points are not "treated the same," but symbolic execution is used to cover all points in a class so selection does not matter. Where specification and program classes exactly coincide, points have the significance that a prescriptive specification was followed. Where they do not, the prospect of failures is good. The programmer has not followed the specification, but chosen to do it another way, and that raises the possibility of left-out cases, cases treated incorrectly, etc. Each such possibility is localized in a partition. However, it should be noted that the method performs best when there are not very many differences, when the prescriptive specification was followed except for a few, unrelated deviations. This is probably just the well-known phenomenon that path classes are not helpful in discovering that something is missing from code or specification. For example, the method of [22] does much better on a triangle program that is nearly right than it does on the original version [3] with the "illegal input" logic omitted.

Thus the success of partition testing for debugging turns on nose-rubbing when the specification is unlike the program, and on boundary errors when they are like. In both cases the precision of the class in which a failure is found is helpful in locating the fault.

## 5. Goals and Prospects for Testing Theory

"Absolute and correctness-based" best characterize existing confidence-testing theories. They have shown that reliability and validity cannot be established algorithmically, then investigated restricted cases which can be solved. Perhaps fault-based testing is the most successful such theory. The basis in correctness cannot be abandoned in a confidence theory, but the absolute ideas could be replaced with probabilistic ones.

A confidence theory must at a minimum apply to any particular test situation in which no failures are observed. It can be argued that no confidence whatsoever can be placed in a program with known faults, certainly true if its users are malicious. The situation in which a few obscure failures is tolerated is an important practical one, but the idea of probabilistic correctness cannot apply to it. We therefore should consider only the release situation: software does not fail on any test conducted, and since the testers are at their wits' end, there is nothing to do but let the users have it. The primary goal of confidence theory is therefore to assign a probability of correctness to any release test. This probability must depend only on the test situation, although more information about the test and program may have to be collected than at present. For example, if a theory involves the textual distribution of tests, as suggested in Section 3.2, then that distribution must be measured.

It is possible to perform real experiments to validate a probabilistic confidence testing theory. Software field performance can be compared to predictions of a theory, but there are many confounding factors so great care is required. For example, field failures are mostly misunderstandings, and important as these are for the whole process of software development, they apply to flaws in documentation and specification, not to release testing. Because validation is so difficult, it is important that a testing theory be *plausible*. Plausibility can be gained through negative predictions: there are many opportunities for a theory to properly assign a *low* probability of correctness. For example: a large program or a large specification domain subjected to a small test; a test that fails elementary control or data coverage criteria; tests generated by third parties lacking knowledge of the specification and program; "devil's advocate" tests generated in an attempt to have no significance. All these provide opportunities for a confidence theory to denigrate the test, and a plausible theory would do so.

A secondary goal of confidence-testing theory is the analysis of testing methods. This does not follow from the ability to analyze the release-test situation. It may happen that the analysis is mathematically intractable, in the sense that closed-form descriptions cannot be obtained for all tests that follow a given method. It can happen that two methods cannot be compared, because the intra-method variability is larger than the difference between them. If the arguments in Section 2 are correct, existing methods *will* be difficult to compare, and will not establish a high degree of confidence. Such an intuitively correct result would be a plausibility argument for the theory that can obtain it. As another example of probabilistic theories giving the right kind of results, an analysis of partition testing *vs.* random testing [27] shows that even under conditions favoring partition testing, it does not increase confidence in the correctness of the program. This agrees with the analysis of Section 4 that partition testing has no confidence component.

Even if there were a plausible confidence-testing theory, it would be important to have theories that explain the strengths and weaknesses of debug-testing methods. The primary goal of debug-testing theory is to connect the human programming process, and the faults it introduces into programs, with the process of looking for those faults. Perhaps such a theory could be used to eliminate human mistakes, but it is better to build tools that detect them. Good programming-language compilers use just this idea. Although it might be possible to analyze syntax errors and through training and discipline eliminate them at the human source, it is much better to just make sure they are caught by the compiler. Then the programmer can concentrate on other things, knowing that blunders will be caught. A debugging technique might be based on confidence-testing theory, as suggested in Section 1, but the theory would give no guidance beyond the information that a successful test of a buggy program predicts a low probability of correctness. In contrast, an exercise method like branch testing can be argued to be useful because (for example) it forces tests in which loop code is executed zero times, and programmers have been known to ignore this case.

In the confusion of debug- and confidence-testing theory, most of the important questions about debugging have not been asked. For example, a large system might be constructed as a collection of modules using the technique of information hiding. What kinds of mistakes do programmers tend to make when following this method? What kind of tests will expose the faults? Can those tests be conducted module-by-module, or do they necessarily involve the whole system? The answers to such questions are an important part of a debug-testing theory of development using information hiding.

**Acknowledgements**

Dave Preston and David Barton found several mistakes and confusions in this paper when they read it for a graduate seminar.

## References

1. G. Myers, *The Art of Software Testing*, Wiley, 1979.

2. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, On the automated generation of program test data, *IEEE Trans. Software. Eng.* SE-2 (Dec., 1976), 293-300.

3. E. F. Miller and R. A. Melton, Automated generation of testcase datasets, 1975 Int. Conf. Reliable Software, Los Angeles, 1975.

4. R. DeMillo, R. Lipton, & F. Sayward, Hints on test data selection: help for the practicing programmer, *Computer* 11 (April, 1978), 34-43.

5. R. Hamlet, Testing programs with the aid of a compiler, *IEEE Trans. Software Eng.* SE-3 (1977), 279-290.

6. S. Rapps and E. J. Weyuker, Data flow analysis techniques for test data selection, *Proc. 6th ICSE*, Tokyo, 1982, 272-278.

7. M. D. Weiser, J. D. Gannon, and P. R. McMullin, Comparison of test coverage metrics, submitted for publication.

8. M. R. Woodward, M. A. Hennell, and D. Hedley, A measure of control flow complexity in program text, *IEEE Trans. Software. Eng.* SE-5 (Jan., 1979), 45-50.

9. W. Howden, Reliability of the path analysis testing strategy, *IEEE Trans. Software Eng.* SE-2(1976), 208-215.

10. H. Mills, The new math of computer programming, *CACM* 18 (Jan., 1975), 43-48.

11. R. Hamlet and H. Mills, Functional Semantics, University of Maryland TR-1129, Feb., 1983.

12. R. Hamlet, Testing programs with finite sets of data, *The Comp. J* 20 (Aug., 1977), 232-237.

13. E. Weyuker, An error-based testing strategy, New York University TR 027, Jan., 1981.

14. R. Hamlet, Test reliability and software maintenance, *Proc. COMPSAC 78*, Chicago, Nov., 1978, 315-320.

15. O-J. Dahl, E. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

16. J. Goodenough and S. Gerhart, Toward a theory of test data selection, *Proc Int. Conf. on Reliable Software*, Los Angeles, 1975, 493-510.

17. E. Weyuker and T. Ostrand, Theories of program testing and the application of revealing subdomains, *IEEE Trans. Software Eng.* SE-6(1980), 236-246.

18. M. Geller, Test data as an aid in proving program correctness, *CACM* 21 (May, 1978), 368-375.

19. R. Hamlet, Theoretical issues in software engineering TR 82/8 Department of Computer Science, University of Melbourne, Parkville, September, 1982.

21. D. Richardson and L. Clarke, A partition analysis method to increase program reliability, *Proc. 5th Int. Conf. on Software Engineering*, San Diego, 1981, 244-253.

22. D. J. Richardson and L. A. Clarke, On the effectiveness of the partion analysis method, Workshop on Effectiveness of Testing and Proving Methods, Avalon, CA, 1982.

23. J. Gannon et al., Data abstraction implementation, specification, and testing, *TOPLAS* 3 (July, 1981), 211-223.

24. L. Morell, A Theory of Error-based Testing, Ph.D. thesis, Department of Computer Science, University of Maryland, 1983.

25. S. J. Zeil, Perturbation testing for computation errors, *Proc. 7th ICSE*, Orlando, FL, 1984, 257-265.

26. E. J. Weyuker, The applicability of program schema results to programs, *Int. J. Computer and Information Sci.* 8, 387-403.

27. J. Duran and S. Ntafos, A report on random testing, *Proc. 5th Int. Conf. on Software*

*Engineering*, San Diego, 1981, 179-183.

28. M. V. Zelkowitz *et al.*, Case studies of software engineering practices in the US and Japan, to appear in *Computer*.

29. J. Duran and J. Wiorkowski, Toward models for probabilistic program correctness, *Proc. ACM Software Quality & Assurance Workshop*, San Diego, 1978.

30. J. D. Gannon, P. R. McMullin, and R. G. Hamlet, Data abstraction implementation, specification, and testing, *TOPLAS* 3 (July, 1981), 211-223.